

# Asynchronous Exclusive Selection \*

Bogdan S. Chlebus <sup>†</sup>

Dariusz R. Kowalski <sup>‡</sup>

## Abstract

An asynchronous system of  $n$  processes prone to crashes, along with a number of shared read-write registers, is the distributed setting. We consider assigning integer numbers to processes in an exclusive manner, in the sense that no integer is assigned to two distinct processes. In the problem called Renaming, any  $k \leq n$  processes, which hold original names from a range  $[N] = \{1, \dots, N\}$ , contend to acquire unique integers in a smaller range  $[M]$  as new names using some  $r$  auxiliary shared registers. We develop a wait-free  $(k, N)$ -renaming solution operating in  $\mathcal{O}(\log k(\log N + \log k \log \log N))$  local steps, for  $M = \mathcal{O}(k)$ , and with  $r = \mathcal{O}(k \log(N/k))$  auxiliary registers. Our approach is based on having processes traverse paths in graphs with suitable expansion properties, where names represent nodes and processes compete for the name of each visited node. We develop a wait-free  $k$ -renaming algorithm operating in  $\mathcal{O}(k)$  time, with  $M = 2k - 1$  and with  $r = \mathcal{O}(k^2)$  registers. We develop an almost adaptive wait-free  $N$ -renaming algorithm, with  $N$  known, operating in  $\mathcal{O}(\log^2 k(\log N + \log k \log \log N))$  time, with  $M = \mathcal{O}(k)$  and with  $r = \mathcal{O}(n \log(N/n))$  registers. We give a fully adaptive solution of Renaming, with neither  $k$  nor  $N$  known, having  $M = 8k - \lg k - 1$  as the bound on new names, operating in  $\mathcal{O}(k)$  steps and using  $\mathcal{O}(n^2)$  registers. As regards lower bounds, we show that a wait-free solution of Renaming requires  $1 + \min\{k - 2, \log_{2r} \frac{N}{2M}\}$  steps in the worst case. We apply renaming algorithms to obtain solutions to a problem called Store&Collect, which is about operations of storing and collecting under additional requirements. In particular, when both  $k$  and  $N$  are known, then storing can be performed in  $\mathcal{O}(\log k(\log N + \log k \log \log N))$  steps and collecting in  $\mathcal{O}(k)$  steps, with  $r = \mathcal{O}(k \log(N/k))$  registers. Additionally, when  $N = \text{poly}(n)$  is known but  $k$  is not, then storing takes  $\mathcal{O}(\log^2 k(\log n + \log k \log \log n))$  steps and collecting  $\mathcal{O}(k)$  steps, with  $\mathcal{O}(n \log n)$  registers. We consider the problem called Unbounded-Naming in which processes may repeatedly request new names, while no name can be reused once assigned, so that infinitely many integers need to be exclusively assigned as names in an execution. In such circumstances, for no fixed integer  $i$  can one guarantee in a wait-free manner that  $i$  is eventually assigned to be a name, so some integers may never be used; the upper bound on the number of such unused integers is used as a measure of quality of a solution. We show that Unbounded-Naming is solvable in a non-blocking way such that at most  $n - 1$  nonnegative integers are never assigned as names, which is best possible, and in a wait-free manner such that at most  $n(n - 1)$  nonnegative integers are never assigned as names.

**Keywords:** asynchrony, crashes, read-write shared registers, renaming, store and collect, non-blockiness, wait-freeness, graph expansion.

---

\* A preliminary version of this paper appeared as [30].

<sup>†</sup>Department of Computer Science and Engineering, University of Colorado Denver, Denver, Colorado 80217, USA.  
Work supported by the National Science Foundation under Grant No. 1016847.

<sup>‡</sup>Department of Computer Science, University of Liverpool, Liverpool L69 3BX, UK.

# 1 Introduction

We consider asynchronous distributed systems consisting of some  $n$  processes prone to crashes that use read-write registers for inter-process communication. The problems we study regard assigning nonnegative integer values to the processes in an *exclusive* fashion, which means that no integer is assigned to two distinct processes. We seek solutions with non-blocking properties, preferably wait-free, see [21, 36, 38]. When an integer  $i$  is assigned to a process  $p$  exclusively, then we say that  $i$  is  $p$ 's *name*. This does not mean that  $p$  will have at most one name at a time, as in many “naming” and “renaming” problems, and we may even allow a process to accumulate an infinite number of such names in an infinite execution.

In the problem Renaming, a set of  $k \leq n$  processes, each equipped with its original name from a large range  $[N] = \{1, \dots, N\}$ , contend to acquire unique integers in a smaller range  $[M]$  as new names, using some  $r$  shared registers. When an algorithm can have some of these parameters as a part of code, then we indicate this by attaching these parameters to the name of the problem and its solutions. For instance,  $(k, N)$ -*renaming* algorithm is for the case when both  $k$  and  $N$  are known and set in code, while  $M$  and  $r$  and the time complexity are characteristics of the algorithm, which thus are functions of  $k$  and  $N$ , and possibly also of  $n$ . A  $k$ -*renaming* algorithm is to work for any range  $[N]$  of the original names and for up to  $k$  contending processes, while a  $N$ -*renaming* algorithm is to handle the original names in the range  $[N]$  while the contention  $k \leq n$  is arbitrary. A *(fully) adaptive* renaming algorithm is to work for any contention  $k \leq n$ , which is not a part of code, while the time complexity is measured as a function of  $k$ . An adaptive  $k$ -renaming algorithm is to work for any range  $[N]$  of the original names, since it is not known in advance, while  $M$  is to be a function of  $k$ , and  $r$  is a function of  $n$ , as this is the maximum value of any  $k \leq n$ . A renaming algorithm is *almost adaptive* if  $N$  is known while  $k$  is not, and both time complexity and the magnitude of  $M$  are functions of  $k$ .

We restrict our attention to *one-time* renaming problems in which each contending process needs to acquire a name from the very beginning of an execution, and no name is ever released to be possibly reused. Time complexity is measure by *local steps*, which is the maximum number of steps a process takes before achieving a state required by the problem.

In the problem Store&Collect, some  $k$  processes perform two operations **Store** and **Collect**. The **Store** operation by a process  $p$  proposes a value, and **Collect** results in learning all values proposed by processes, one value per process.

The problem Unbounded-Naming is about processes that repeatedly require new names, while no name can be reused once already assigned to, so that infinitely many integers need to be exclusively assigned as names. To relate this to the previous work on models with infinite arrivals of processes and infinitely many shared registers, see [13, 34, 40]; our model assumes finitely many processes but infinitely many registers. For no fixed integer  $i$  can one guarantee in a wait-free manner that  $i$  is eventually assigned to be a name in an instance of Unbounded-Naming, so some integers may never be used. An upper bound on the number of such unused integers is proposed as a measure of quality of a solution for Unbounded-Naming. We consider the problem to implement a *repository* of values in infinitely many read-write registers. The values are generated in a dynamic fashion. A value has been deposited in a register when it is stored in this register and will never be overwritten. In this problem we want each register to be eventually used to deposit some value. The problem is an illustration of applicability of unbounded naming, as names can be used to identify register to make deposits.

**Contributions of this paper.** We now overview the results in greater detail. The main underlying idea in renaming algorithms is to have processes traverse paths in graphs with suitable expansion properties, with nodes representing names and processes competing for the name of each visited node. Previously known approaches often relied on the underlying graphs of a regular topology, like grids.

We develop a wait-free  $(k, N)$ -renaming algorithm operating in  $\mathcal{O}(\log k(\log N + \log k \log \log N))$  local steps, for  $M = \mathcal{O}(k)$ , and with  $r = \mathcal{O}(k \log(N/k))$  auxiliary registers. This is the first deterministic algorithm known to have a sublinear local step performance for sub-exponential range of  $N$ , and a polylogarithmic step complexity for a polynomial range of  $N$ , with  $N$  considered as a function of  $k$ .

We show that  $1 + \min\{k - 2, \log_{2r} \frac{N}{2M}\}$  local steps are required in the worst case by any wait-free  $(k, N)$ -renaming algorithm that assigns names from the range  $[M]$  and uses  $r$  registers. This is a first known lower bound on the local-step time complexity of Renaming that involves all the four parameters. In particular, when  $N$  is unknown and hence could be arbitrarily large, while  $M$  is bounded, say, as a function of  $k$ , then  $k - 1$  is the lower bound; this resembles the lower bounds given by Jayanti et al. [39]. Recently, an  $\Omega(k)$  lower bound on time, under additional assumptions, was proven by Alistarh et al. [9] by a different argument.

We develop a wait-free  $k$ -renaming algorithm operating in  $\mathcal{O}(k)$  time, with  $M = 2k - 1$  and with  $r = \mathcal{O}(k^2)$  auxiliary registers. The time complexity of this algorithm is asymptotically optimal, by the lower bound we show and the fact that the algorithm works for arbitrary  $N$ . The value  $M = 2k - 1$  is known to be best possible size of a range of names [20, 37] for shared read-write registers. Ours is the first algorithm known that has simultaneously the following two properties: the time complexity is  $\mathcal{O}(k)$  and  $M = \mathcal{O}(k)$ . Among the previously known algorithms that run in time  $\mathcal{O}(k)$ , the value  $M = k(k + 1)/2$  was smallest known; it is achieved by an algorithm of Moir and Anderson [41]. The fastest algorithm known before among those having  $M = \mathcal{O}(k)$  was given by Attiya and Fournier [16], it operates in  $\mathcal{O}(k \log k)$  time. The fastest algorithm known prior to this work and with  $M = 2k - 1$  as the bound on new names runs in time  $\mathcal{O}(k^2)$ , it was given by Afek and Merritt [5].

We develop an almost adaptive wait-free  $N$ -renaming algorithm of step complexity that is  $\mathcal{O}(\log^2 k(\log N + \log k \log \log N))$ , for unknown contention  $k$ , with  $M = \mathcal{O}(k)$  and with  $r = \mathcal{O}(n \log(N/n))$  registers. Attiya and Fournier [16] gave an algorithm working in time  $\mathcal{O}(N)$  with  $M = 2k - 1$  as the bound on new names.

We give a fully adaptive renaming algorithm, with neither  $k$  nor  $N$  known, having  $M = 8k - \lg k - 1$  as the bound on new names, operating in  $\mathcal{O}(k)$  local steps and using  $\mathcal{O}(n^2)$  auxiliary registers. This is an improvement with respect to time performance over the previously known algorithms. The algorithm of Moir and Anderson [41] works in time  $\mathcal{O}(k)$  with  $M = \mathcal{O}(k^2)$  using  $\mathcal{O}(n^2)$  registers. The algorithm of Attiya and Fournier [16] operates in  $\mathcal{O}(k \log k)$  time with  $M = \mathcal{O}(k)$ . The algorithm of Afek and Merritt [5] runs in time  $\mathcal{O}(k^2)$  with  $M = 2k - 1$ .

We apply renaming algorithms to obtain solutions to Store&Collect. When both  $k$  and  $N$  are known, then storing can be performed in  $\mathcal{O}(\log k(\log N + \log k \log \log N))$  local steps while collecting in  $\mathcal{O}(k)$  local steps, for the number of registers  $r = \mathcal{O}(k \log(N/k))$ . When  $N = \mathcal{O}(n)$  is known but  $k$  is not, then storing can be accomplished in  $\mathcal{O}(\log^2 k(\log n + \log k \log \log n))$  local steps, while collecting in  $\mathcal{O}(k)$  local steps, for  $r = \mathcal{O}(n)$ . When  $N = \text{poly}(n)$  is known but  $k$  is not, then storing can be done in  $\mathcal{O}(\log^2 k(\log n + \log k \log \log n))$  local steps, while collecting

in  $\mathcal{O}(k)$  local steps, for  $r = \mathcal{O}(n \log n)$ . Afek and De Levie [4] gave an adaptive solution, when neither  $k$  nor  $N$  is known, achieving storing in  $\mathcal{O}(k)$  local steps and collecting in  $\mathcal{O}(k)$  local steps, for  $r = \mathcal{O}(n^2)$ ; this result follows directly from our adaptive solution of Renaming. Our deterministic algorithm also improves the step complexity of storing in their solution for  $N = \mathcal{O}(n)$  from  $\mathcal{O}(k)$  to  $\mathcal{O}(\log^2 k (\log n + \log k \log \log n))$ , while the remaining performance metrics stay the same. Attiya et al. [19] developed a randomized Store&Collect solution in which storing takes time  $\mathcal{O}(\log k \log \log k)$  and collecting takes time  $\mathcal{O}(k)$ , for  $N = \mathcal{O}(n)$  and  $r = \mathcal{O}(n)$ . Our deterministic solution has the time of storing within the factor of  $\log k (\log n + \log k \log \log n) / \log \log k$  from their expected time complexity.

We consider a problem called Unbounded-Naming which is about processes working to claim nonnegative integers in an exclusive manner as names. We show that Unbounded-Naming is solvable in a non-blocking way so that at most  $n - 1$  integers are never assigned as names, which is best possible, and in a wait-free manner so that at most  $n(n - 1)$  values are never assigned as names. Problem Unbounded-Naming has not been considered before in the literature known to the authors of this paper.

**Previous work.** Now we describe the context of this work by reviewing previous research on renaming. We restrict our attention to asynchronous systems with shared memory consisting of only read-write registers; for a comprehensive survey of renaming see Alistarh [8].

The problem of renaming was introduced by Attiya et al. [14] in the model of asynchronous message-passing. They showed that  $n$  processes may assign themselves new names in the range  $[n + f]$ , where  $f < n$  is an upper bound on the number of crashes. This was given as an instance of a non-trivial algorithmic problem with a wait-free solution for environments in which Consensus cannot be solved; see [21, 33]. The range  $[M = n + f]$  was shown to be smallest possible by Herlihy and Shavit [37] and Attiya and Rajsbaum [20]. Next we consider a scenario when there are some arbitrary  $k \leq n$  contending processes with their original names in some large range  $[N]$  that want to obtain new names in some small range  $[M]$ . Borowsky and Gafni [22] gave a wait-free algorithm with time complexity  $\mathcal{O}(k^2 N)$  for  $M = 2k - 1$ . Moir and Anderson [41] gave a solution with time complexity  $\mathcal{O}(k)$ , for new names of magnitude  $M = k(k + 1)/2$  and using  $r = \mathcal{O}(k^2)$  registers. Attiya and Fournier [16] gave an algorithm working in time  $\mathcal{O}(k \log k)$  for  $M = 6k - 1$ , and another of time complexity  $\mathcal{O}(N)$  for  $M = 2k - 1$ . Afek and Merritt [5] developed an algorithm working in time  $\mathcal{O}(k^2)$  for  $M = 2k - 1$ .

A renaming algorithm is *long-lived* when processes invoke the operations to request a name and to release the current name, subject to the constraint that exclusiveness of a name needs to hold within the interval from acquiring to releasing the name. It is assumed that at most  $k$  processes contend for names concurrently. The following is a selection of published long-lived renaming algorithms. Burns and Peterson [25] gave a solution of time complexity  $\mathcal{O}(Nk^2)$ , for  $M = 2k - 1$  and  $r = \mathcal{O}(N^2)$ . Moir and Anderson [41] improved the time to  $\mathcal{O}(Nk)$ , for  $M = k(k + 1)/2$  and  $r = \mathcal{O}(Nk^2)$ . Further improvements were due to Buhrman et al. [24], who achieved  $\mathcal{O}(k^3)$  time, for  $M = k(k + 1)/2$  and the number of registers  $r = \mathcal{O}(k^4 \min\{3^k, N\})$ , and to Moir and Garay [42] whose algorithm achieved  $\mathcal{O}(k^2)$  time complexity, for  $M = k(k + 1)/2$  and  $r = \mathcal{O}(k^3)$  registers, and who also gave another solution with  $\mathcal{O}(k^4)$  time, for  $M = 2k - 1$  and  $r = \mathcal{O}(k^4)$ . For other work, see the papers by Afek et al. [2, 3, 7], Brodsky et al. [23], and Eberly et al. [32].

Randomized renaming algorithms were given by Alistarh et al. [10, 11, 12]. Lower bounds on renaming were given by Alistarh et al. [9], Castañeda et al. [27], Castañeda and Rajsbaum [28, 29],

---

**Procedure** COMPETE-FOR-REGISTER ( $R$ )

---

```
read: contentionp ←  $H_R$ 
if contentionp = null then write  $H_R$  ←  $p$  else exit
read: contentionp ←  $R$ 
if contentionp = null then write  $R$  ←  $p$  else exit
read: contentionp ←  $H_R$ 
if contentionp =  $p$  then win else exit
```

---

Figure 1: A pseudocode of a procedure to win a register  $R$ . Code for a process  $p$ , which uses a local variable `contentionp`. Shared registers  $R$  and  $H_R$  are initialized to null.

and Helmi et al. [35]. Previous work on collecting algorithms includes papers by Afek [6], Attiya et al. [15, 17, 18], Chlebus et al. [31] and Saks et al. [43].

## 2 Technical preliminaries

We assume an asynchronous system with some  $n$  processes prone to crashes and a set of read-write registers. Each process is identified by its *original name* which is a unique number in the interval  $[N] = \{1, \dots, N\}$ .

The following is the standard terminology regarding delays of enabled operations; see [21, 38]. When, for any configuration in an execution, some process will eventually complete an invoked operation, then the executed algorithm is *non-blocking*. When, for any configuration in an execution, each process will eventually complete an invoked operation, even when all the remaining processes have crashed, then the algorithm is *wait-free*.

**Competing for registers.** We will use a procedure to compete for a shared register, which needs to have the following properties:

Wins are guaranteed with no contention: If there is exactly one process  $p$  working to win  $R$ , then  $p$  eventually wins  $R$ .

Wins are exclusive: If some contender wins  $R$ , then no other contender will ever win  $R$ .

Observe that this specification does not require a register to be won by a process when there are multiple contenders but also does not exclude such a possibility. To implement a competition for a register  $R$ , we use an auxiliary dedicated shared register  $H_R$  initialized as `null`. This register  $H_R$  is used as a placeholder to store a reservation for  $R$ . A pseudocode of a procedure for  $p$  is given in Figure 1.

**Lemma 1** *Procedure  $\text{COMPETE-FOR-REGISTER}_p()$  is an implementation of competition for a register.*

**Proof:** To show correctness, consider two cases corresponding to the specification. If there is only one process  $p$  contending to win a register  $R$ , then  $p$  will eventually have written value  $p$  to both  $H_R$  and  $R$ , so that the last read from  $H_R$  makes the process  $p$  a winner. Suppose it is otherwise, in that some process  $p$  wins  $R$  and there is another contender  $q$ . If the first read of  $H_R$  by  $q$  does not return **null** then  $q$  exits immediately. When this is not the case, then this means that process  $q$  read from register  $H_R$  before process  $p$  wrote to  $H_R$ . What happened after this read was that process  $p$  managed to write to  $H_R$  and then to  $R$  and then check that the value  $p$  was still in register  $H_R$ , as all this is required to win  $R$ . So the value  $p$  at register  $H_R$  is overwritten only when register  $R$  already stores the value  $p$ . Now, the next read of register  $R$  by process  $q$  returns the value  $p \neq q$ , so process  $q$  exits.  $\square$

**Graphs.** Let  $G = (V, W, E)$  be a simple bipartite graph, with the nodes partitioned into the set of *inputs*  $V$  and the set of *outputs*  $W$ , and  $E$  as the set of edges. We say that graph  $G$  has *input-degree*  $\Delta$  if each node in  $V$  is connected to exactly  $\Delta$  neighbors in  $W$ . A node  $v \in W$  is a *unique neighbor* of set  $S \subseteq V$  if  $v$  is adjacent to exactly one node in  $S$ . Graph  $G$  is said to be an  $(L, \Delta, \varepsilon)$ -*lossless-expander* if  $\Delta$  is the input-degree of  $G$  and every subset  $X$  of  $V$  of size  $|X| \leq L$  has more than  $(1 - \varepsilon)|X|\Delta$  neighbors in  $W$ .

**Lemma 2** *If a bipartite graph  $G = (V, W, E)$  is an  $(L, \Delta, \varepsilon)$ -lossless-expander, for some parameters  $L$  and  $\varepsilon < 1/2$ , then for every set  $X \subseteq V$  of size  $|X| \leq L$  there is a partial matching in  $G$ , between the nodes in  $X$  and the unique-neighbors of  $X$ , that has more than  $(1 - 2\varepsilon)|X|$  edges.*

**Proof:** If  $G$  is a  $(L, \Delta, \varepsilon)$ -lossless-expander then more than  $(1 - 2\varepsilon)\Delta|X|$  nodes among the neighbors of  $X$  are unique neighbors, by Lemma 1.1 in [26]. We can match these inputs to their unique neighbors.  $\square$

We use  $\lg x$  to denote the logarithm of  $x$  to the base 2. We will resort to the existence of lossless expanders with the following properties:

**Lemma 3** *Given two disjoint sets  $V$  and  $W$ , such that  $|W| = 12e^4 L \lg \frac{|V|}{L}$  for some parameter  $L$  such that  $1 \leq L \leq \frac{|V|}{200e^4}$ , there exists a bipartite graph  $G = (V, W, E)$  of input-degree  $\Delta = 4 \lg \frac{|V|}{L}$  that is a  $(L, \Delta, \frac{1}{4})$ -lossless-expander.*

**Proof:** We show that a randomly selected set of edges between the nodes in  $V$  and  $W$  meets the requirements. More precisely, for each node  $v \in V$ , select uniformly at random  $\Delta$  neighbors in  $W$ , where the value of  $\Delta$  will be defined later. We will show that the resulting graph is a  $(L, \Delta, \frac{1}{4})$ -lossless-expander with a probability greater than 0.

Fix a subset  $X \subseteq V$  of size  $x = |X| \leq L$ , and a subset  $Y \subseteq W$  of the size  $\frac{3}{4}x\Delta$ . The probability that all the neighbors of  $X$  are in the set  $Y$  is at most

$$\left( \frac{\binom{|Y|}{\Delta}}{\binom{|W|}{\Delta}} \right)^x \leq \left( \frac{|Y|e}{|W|} \right)^{x\Delta} \leq \left( \frac{\frac{3e}{4} \cdot x\Delta}{|W|} \right)^{x\Delta}.$$

Observe that there are

$$\binom{|V|}{x} \leq \left(\frac{|V|e}{x}\right)^x$$

different subsets  $X \subseteq V$  of size  $x$ , and that there are

$$\binom{|W|}{\frac{3}{4}x\Delta} \leq \left(\frac{|W|e}{\frac{3}{4}x\Delta}\right)^{3x\Delta/4}$$

different subsets  $Y \subseteq W$  of size  $\frac{3}{4}x\Delta$ . Therefore, the probability that for a given value of size  $x$  there exists a set  $X \subseteq V$  with at most  $\frac{3}{4}x\Delta$  neighbors is at most

$$\begin{aligned} \left(\frac{|V|e}{x}\right)^x \cdot \left(\frac{|W|e}{\frac{3}{4}x\Delta}\right)^{3x\Delta/4} \cdot \left(\frac{\frac{3e}{4}x\Delta}{|W|}\right)^{x\Delta} &\leq \left(\frac{|V|e}{x}\right)^x \cdot \left(\frac{\frac{3e^3}{4}x\Delta}{|W|}\right)^{x\Delta/4} \\ &\leq \left(\frac{|V|}{L} \cdot \frac{L}{x} \cdot \left(\frac{\frac{3e^4}{4}x\Delta}{|W|}\right)^{\Delta/4}\right)^x \\ &\leq \left(\frac{|V|}{L} \cdot \frac{L}{x} \cdot \left(\frac{1}{4} \cdot \frac{x}{L}\right)^{\lg \frac{|V|}{L}}\right)^x, \end{aligned}$$

which in turn is at most  $1/2^x$ . The probability that such set  $X \subseteq V$  exists for *any* size  $x \leq L$  is at most  $\sum_{x=1}^L \frac{1}{2^x} < 1$ . By the probabilistic method, a graph with the properties we seek exists.  $\square$

### 3 Bounded selection

We consider a problem called Majority-Renaming, which is to assign new names to at least half of some  $k$  contending processes. An algorithm is  $(k, N)$ -majority-renaming with bound  $M$  on new names if at least half of the  $k$  contending processes with original names in  $[N]$  acquire a unique name in  $[M]$  each, while the numbers  $k$  and  $N$  can be a part of code of the algorithm.

We first find a solution for Majority-Renaming based on lossless expanders with good unique-neighbors properties, then we apply it to develop solutions for Renaming itself. Using a renaming algorithm with the information represented by its parameters, we then argue how to use it to obtain an adaptive solution. Finally, we discuss how to use the obtained renaming algorithms to solve Store&Collect.

**Renaming when both  $k$  and  $N$  are known.** We present an algorithm called MAJORITY( $\ell, N$ ), where  $\ell \leq N/(200e^4)$ , which is  $(\ell, N)$ -majority-renaming, where  $M = 12e^4\ell \lg(N/\ell)$  as a bound on new names.

Let  $G$  be a bipartite graph as in Lemma 3, which is to be a part of code of the algorithm. Let  $V = [N]$ ,  $W = [M]$ , and  $\Delta$  denote the input-degree of graph  $G$ . The set  $V$  corresponds to all possible names of processes, the set  $M$  contains the set of pairs of registers, a pair representing a name, while the edges define which registers will be competed for by each process.

The process of competition to win registers in algorithm MAJORITY( $\ell, N$ ) proceeds as follows. A process  $p \in [N]$ , corresponding to a node  $p$  in  $V$ , attempts to win a register corresponding to its first neighbor in  $W$ , using procedure COMPETE-FOR-REGISTER given in Figure 1, and if  $p$  fails then it attempts to win the register of its second neighbor in  $W$ , and so on. If  $p$  fails all the  $\Delta$

competitions, then  $p$  stops, otherwise  $p$  adopts the number of the captured register in  $W$  as its new name.

**Lemma 4** *Algorithm MAJORITY( $\ell, N$ ), for  $\ell \leq N/(200e^4)$  is  $(\ell, N)$ -majority-renaming with  $M = 12e^4\ell \lg(N/\ell)$  as a bound on new names. It operates in  $\mathcal{O}(\log N)$  local steps and uses  $\mathcal{O}(M)$  auxiliary registers.*

**Proof:** It follows directly from Lemma 2 applied to graph  $G$  that a majority of contending processes have unique neighbors not shared with other active processes, since by its definition the graph  $G$  is a  $(\ell, \Delta, 1/4)$ -lossless-expander as stated in Lemma 3. We observe that if an active process has a unique neighbor then it eventually wins some register representing its neighbor, by Lemma 1. Hence a majority of active processes get unique names. The time complexity is proportional to the degree of graph  $G$ , which is  $\mathcal{O}(\log N)$ , while the number of registers is proportional to  $M$ , as we use two registers per node in  $W$ .  $\square$

Next we consider an algorithm called BASIC-RENAME( $k, N$ ), which is  $(k, N)$ -renaming with  $M = 24e^4k \lg(N/k)$  as a bound on new names. It proceeds through  $\lg k + 1$  stages. A process  $p \in [N]$  executes the stages until it gets a unique name. In a stage  $i$ , where  $0 \leq i \leq \lg k$ , process  $p$  executes MAJORITY( $k/2^i, N$ ) on the set of pairs of registers  $M_i$ , where  $|M_i| = 12e^4(k/2^i) \lg \frac{N}{k/2^i}$ . We assume that the sets  $M_i$  are mutually disjoint. The union of all these sets  $M_i$  constitute a collection of new names.

**Lemma 5** *Algorithm BASIC-RENAME( $k, N$ ) is  $(k, N)$ -renaming with  $M = 24e^4k \lg(N/k)$  as a bound on new names. It operates in  $\mathcal{O}(\log k \log N)$  local steps and uses  $\mathcal{O}(k \log(N/k))$  auxiliary registers.*

**Proof:** By Lemma 4, the calls of all the procedures MAJORITY( $k/2^i, N$ ) guarantee a geometric progress, with a factor of at least  $1/2$  contributed by a stage, in getting new names. This continues until there remain at most one process without a new name, which then eventually also gets a name. This takes  $\mathcal{O}(\log k \cdot \log N)$  local steps in total, by Lemma 4. The number of new names is bounded above by

$$\sum_{i=0}^{\lg k} 12e^4(k/2^i) \lg(N2^i/k) \leq 24e^4k \log(N/k) ,$$

which is also asymptotically a number of shared registers.  $\square$

The next algorithm POLYLOG-RENAME( $k, N$ ) is  $(k, N)$ -renaming with  $M = 768e^4k$  as a bound on new names. It proceeds through a sequence of epochs. A process  $p \in [N]$  executes consecutive epochs numbered by  $j$ , for  $j \geq 1$ , in each one getting a new name. This continues as long as the upper bound on the range of the new names, determined by the properties of an epoch, is at most  $M$ . A process acquires its ultimate name during the last such an epoch. In epoch  $j$ , process  $p$  executes BASIC-RENAME( $k, N_j$ ), where  $N_1 = N$  and  $N_{j+1} = 24e^4k \lg(N_j/k)$  is a bound on new names of BASIC-RENAME( $k, N_j$ ). All these executions of instantiations of algorithm BASIC-RENAME are on disjoint sets of auxiliary registers.

**Theorem 1** *Algorithm POLYLOG-RENAME( $k, N$ ) is  $(k, N)$ -renaming with  $M = 768e^4k$  as a bound on new names. It operates in  $\mathcal{O}(\log k(\log N + \log k \log \log N))$  local steps and with  $\mathcal{O}(k \log(N/k))$  auxiliary registers.*



**Proof:** By Lemma 5, each epoch results in a suitable shrinking of the range of names. More precisely, we have  $N_1 = N$ , then  $N_2 = 24e^4 k \lg(N/k)$ , and, for any index  $j$ , the ratio  $N_{j+1}/N_j$  can be estimated from above as follows:

$$\begin{aligned} \frac{24e^4 k \lg(N_j/k)}{24e^4 k \lg(N_{j-1}/k)} &= \frac{\lg(24e^4 \lg(N_{j-1}/k))}{\lg(N_{j-1}/k)} \\ &= \frac{\lg(24e^4)}{\lg(N_{j-1}/k)} + \frac{\lg \lg(N_{j-1}/k)}{\lg(N_{j-1}/k)} \\ &\leq \frac{2}{3} + \frac{5}{32} = \frac{27}{32}, \end{aligned}$$

if only  $N_{j-1} > 768e^4 k$ . Thus after at most the following number of epochs

$$2 + \lg_{32/27} \left( \frac{N_2}{768e^4 k} \right) \leq \mathcal{O}(\log \log N)$$

we obtain a set  $N_{j^*}$  such that  $24e^4 k \leq N_{j^*} \leq 768e^4 k$ . The number of registers needed is  $\mathcal{O}(k \log(N/k))$ , which dominates  $\mathcal{O}(N_2)$ . It is contributed by the first iteration of BASIC-RENAME. The following identities

$$\sum_{j=2}^{j^*} N_j = \mathcal{O}(N_2) = \mathcal{O}(k \log(N/k))$$

follow from the fact that  $N_{j+1}/N_j \leq 27/32$  for every  $j < j^*$ . The number of local steps can be bounded from above as follows:

$$\mathcal{O}(\log k \log N + \sum_{j=2}^{j^*} \log k \log N_j) \leq \mathcal{O}(\log k \cdot \log N + \log^2 k \cdot \log \log N),$$

by Lemma 5 and by bound  $\mathcal{O}(\log \log N)$  on the number of epochs  $j^*$ . □

**Renaming when only  $k$  is known while  $N \geq k$  is not.** We begin by elaborating more on the case when both  $k$  and  $N$  are known.

Let  $\text{MA}(k)$  be an adaptive algorithm given by Moir and Anderson [41] which is  $k$ -renaming with  $M = \mathcal{O}(k^2)$  as a bound on new names, which operates in  $\mathcal{O}(k)$  local steps and uses  $\mathcal{O}(k^2)$  auxiliary registers.

Let  $\text{AF}(k, N)$  be the algorithm of Attiya and Fouren [16] that is  $(k, N)$ -renaming with  $2k - 1$  as a bound on new names, which operates in  $\mathcal{O}(N)$  local steps and uses  $\mathcal{O}(N^2)$  auxiliary registers.

We use algorithm POLYLOG-RENAME together with algorithms AF and MA to obtain a new algorithm called EFFICIENT-RENAME( $k$ ), which is  $k$ -renaming with  $2k - 1$  as a bound on new names for any  $k$  and  $N$ . It is specified as follows.

We first run algorithm  $\text{MA}(k)$ , and then we proceed by running  $\text{POLYLOG-RENAME}(k, k^2)$  and next  $\text{AF}(k, M')$ , where  $M' = \mathcal{O}(k)$  is the range of new names obtained after executing  $\text{POLYLOG-RENAME}(k, k^2)$ , see Theorem 1. Processes start an execution of  $\text{POLYLOG-RENAME}(k, k^2)$  with names obtained in the execution of  $\text{MA}(k)$ , and an execution of  $\text{AF}(k, M')$  with names obtained in  $\text{POLYLOG-RENAME}(k, k^2)$ . The sets of registers used in all three parts are to be disjoint. The final new names come from an execution of  $\text{AF}(k, M')$ .

**Theorem 2** *Algorithm EFFICIENT-RENAME( $k$ ) is  $k$ -renaming with  $2k - 1$  as a bound on new names. It operates in  $\mathcal{O}(k)$  local steps and uses  $\mathcal{O}(k^2)$  auxiliary registers.*

**Proof:** Correctness follows from the fact that an execution of any of the three renaming algorithms correctly handles new names as yielded by a preceding execution, in its whole range. The range of names is reduced first to  $k^2$  by algorithm MA, next to  $M' = \mathcal{O}(k)$  by algorithm POLYLOG-RENAME, by Theorem 1, and finally to  $2k - 1$  by algorithm AF. Here, similarly as in the description of EFFICIENT-RENAME( $k$ ), the number  $M' = \mathcal{O}(k)$  stands for a range of new names obtained in the execution of POLYLOG-RENAME( $k, k^2$ ). It follows from Theorem 1, from the properties of MA algorithm in [41], and by the properties of AF algorithm given in [16], that the local step complexity of algorithm EFFICIENT-RENAME( $k$ ) is  $\mathcal{O}(k + \log^2 k \log \log k + M') = \mathcal{O}(k)$ . The number of needed registers is at most

$$\mathcal{O}(k^2 + k \log(k^2/k) + (M')^2) = \mathcal{O}(k^2) ,$$

by the respective properties of algorithms MA( $k$ ) and AF( $k, M'$ ), and by Theorem 1. □

**Renaming when only  $N$  is known while  $k$  is not.** We present algorithm called ALMOST-ADAPTIVE( $N$ ), which aims to accomplish renaming for any unknown number  $k$  of contending processes, assigning new names of magnitude  $\mathcal{O}(k)$ , when a bound  $N$  on the original names is known.

The algorithm is specified as follows. A process runs algorithm POLYLOG-RENAME( $2^j, N$ ) through consecutive integers  $0 \leq i \leq \lg n$  until it obtains a new name. Such consecutive executions are on disjoint sets of registers and sets of names, and a range of new names for an execution POLYLOG-RENAME( $2^j, N$ ) is to be taken as the first interval of new names not used before.

**Theorem 3** *Algorithm ALMOST-ADAPTIVE( $N$ ) is  $N$ -renaming, where the original names are in  $[N]$ , the value of  $k$  is unknown, and its bound on new names is  $\mathcal{O}(k)$ . It runs in  $\mathcal{O}(\log^2 k (\log N + \log k \log \log N))$  local steps and uses  $\mathcal{O}(n \log(N/n))$  auxiliary registers.*

**Proof:** At most  $k$  processes are still active by the execution POLYLOG-RENAME( $2^i, N$ ) for  $i = \lceil \lg k \rceil$ . Note that indeed the size of a range of new names is  $\mathcal{O}(\sum_{j=0}^i 2^j) = \mathcal{O}(k)$ , by Theorem 1. By the same theorem, the number of steps by the end of executing POLYLOG-RENAME( $2^j, N$ ) for  $i = \lceil \lg k \rceil$  can be bounded as follows

$$\mathcal{O}\left(\sum_{j=0}^i (j \log N + j^2 \log \log N)\right) = \mathcal{O}(\log^2 k (\log N + \log k \log \log N)) .$$

We have the identity

$$\mathcal{O}\left(\sum_{j=0}^i 2^j \log(N/2^j)\right) = \mathcal{O}(n \log(N/n)) ,$$

so indeed  $\mathcal{O}(n \log(N/n))$  registers suffice. □

Theorem 3 can be specialized as follows.

**Corollary 1** *For  $N = \mathcal{O}(n)$ , algorithm ALMOST-ADAPTIVE( $N$ ) is  $N$ -renaming with new names of magnitude  $\mathcal{O}(k)$ , where  $k$  is unknown. It operates in  $\mathcal{O}(\log^2 k (\log n + \log k \log \log n))$  local steps and*

uses  $\mathcal{O}(n)$  auxiliary registers. For  $N = \text{poly}(n)$ , algorithm  $\text{ALMOST-ADAPTIVE}(N)$  is  $N$ -renaming for unknown  $k$  with new names of magnitude  $\mathcal{O}(k)$ . It operates in  $\mathcal{O}(\log^2 k (\log n + \log k \log \log n))$  local steps and uses  $\mathcal{O}(n \log n)$  auxiliary registers.

**Adaptive renaming with both  $k$  and  $N$  unknown.** Now we develop algorithm  $\text{ADAPTIVE-RENAME}$  solving Renaming in a fully adaptive fashion. To this end, we iterate algorithm  $\text{EFFICIENT-RENAME}$  in a “doubling” manner. Specifically, for every integer  $0 \leq i \leq \lg n$ , a process  $p$  executes algorithm  $\text{EFFICIENT-RENAME}(2^i, 2^n)$ , but only until a new name is obtained. Separate executions of  $\text{EFFICIENT-RENAME}$  are on disjoint sets of auxiliary registers and on disjoint sets of names, say, using the next available range of names. Algorithm  $\text{ADAPTIVE-RENAME}$  does not have the parameters  $k$  and  $N$  in its code.

**Theorem 4** *Algorithm  $\text{ADAPTIVE-RENAME}$  solves Renaming in an adaptive way with  $M = 8k - \lg k - 1$  as a bound on new names. It operates in  $\mathcal{O}(k)$  local steps and uses  $\mathcal{O}(n^2)$  auxiliary registers.*

**Proof:** Consider the epoch  $i^* = \lceil \lg k \rceil$  in an execution of  $\text{EFFICIENT-RENAME}$ . At most  $k$  active processes survive until the beginning of their execution of  $\text{EFFICIENT-RENAME}(2^{i^*}, 2^n)$ . By Theorem 2, the magnitude of new names is bounded above by

$$\sum_{i=0}^{i^*} (2^{i+1} - 1) = 2^{i^*+2} - (i^* + 1) \leq 8k - \lg k - 1 .$$

The number of local steps of each process is bounded by

$$\mathcal{O}\left(\sum_{i=1}^{i^*} 2^i\right) = \mathcal{O}(2^{i^*}) = \mathcal{O}(k) ,$$

and the number of auxiliary registers used is bounded by

$$\mathcal{O}\left(\sum_{i=1}^{i^*} 2^{2i}\right) = \mathcal{O}(2^{2i^*}) = \mathcal{O}(n^2) ,$$

again by Theorem 2. Hence  $\mathcal{O}(n^2)$  registers suffice.  $\square$

We observe an alternative solution for Renaming here, which is as follows. First use an adaptive version of algorithm MA, which accomplishes renaming in  $\mathcal{O}(k)$  local steps and  $k^2$  new names using  $\mathcal{O}(n^2)$  registers. This is followed by algorithm  $\text{ALMOST-ADAPTIVE}(k^2)$ . By Theorem 3, the obtained algorithm solves Renaming in  $\mathcal{O}(k + \log^3 k) = \mathcal{O}(k)$  local steps and uses  $\mathcal{O}(n^2 + n \log n) = \mathcal{O}(n^2)$  registers. A drawback of this algorithm is that the range of new names, although still  $\mathcal{O}(k)$ , has a large constant factor in front of  $k$ , which is not the case in  $\text{ADAPTIVE-RENAME}$  algorithm where the range of names is smaller than  $8k - \lg k$ .

**From Renaming to Store&Collect.** We show how to implement store and collect operations in various settings using the renaming algorithms just developed. Specifically, in the first execution of storing, a process first finds its new name, which in renaming algorithms is equivalent to getting a unique register out of some range  $M$  of registers corresponding to the names, and stores its value there. All the subsequently stored values by this process will be deposited there as well. Collect

operation is completed by reading all the registers used in renaming. The number of registers in adaptive renaming solutions depends on  $n$ . Due to that, in order to perform a collect in  $\mathcal{O}(k)$  local steps, we impose an additional structure on the registers corresponding to the names. Namely, we organize them according to their names into consecutive intervals of lengths  $2, 2^2, 2^3, \dots$  and add a control register, initialized to 0, at the beginning of each such an interval. During the first adaptive store operation, a process writes 1 into control registers of each interval encountered before a register to actually store is found. Collecting proceeds by reading the consecutive intervals of registers in order, along with the control registers, until an empty control register is found.

**Theorem 5** *The operations of Store&Collect can be implemented with the following efficiency:*

- (i) *When both  $k$  and  $N$  are known, then storing takes  $\mathcal{O}(\log k(\log N + \log k \log \log N))$  steps and collecting  $\mathcal{O}(k)$  steps, with  $\mathcal{O}(k \log(N/k))$  registers.*
- (ii) *When  $N = \mathcal{O}(n)$  is known but  $k$  is not, then storing takes  $\mathcal{O}(\log^2 k(\log n + \log k \log \log n))$  steps and collecting  $\mathcal{O}(k)$  steps, with  $\mathcal{O}(n)$  registers.*
- (iii) *When  $N = \text{poly}(n)$  is known but  $k$  is not, then storing takes  $\mathcal{O}(\log^2 k(\log n + \log k \log \log n))$  steps and collecting  $\mathcal{O}(k)$  steps, with  $\mathcal{O}(n \log n)$  registers.*
- (iv) *In a fully adaptive solution, without knowing either  $k$  or  $N$ , storing takes  $\mathcal{O}(k)$  steps while collecting takes  $\mathcal{O}(k)$  steps, with  $\mathcal{O}(n^2)$  registers.*

**Proof:** Consider an execution with contention  $k$ . As a renaming subroutine, we use POLYLOG-RENAME( $k, N$ ) in the first case, ALMOST-ADAPTIVE( $N$ ) in the second and third cases, and ADAPTIVE-RENAME in the fourth one. Note that the total number of all registers corresponding to the number of names that are used is  $\mathcal{O}(k)$  in all the four cases, by Theorem 1 for (i), Corollary 1 for (ii) and (iii), and Theorem 4 for (iv). Hence the total number of control registers involved in reading/writing is always  $\mathcal{O}(\log k)$ .

The first store operation of a process takes the following time:  $\mathcal{O}(\log k(\log N + \log k \log \log N))$  local steps in setting (i), by Theorem 1;  $\mathcal{O}(\log^2 k(\log n + \log k \log \log n) + \log k) = \mathcal{O}(\log^2 k(\log n + \log k \log \log n))$  local steps in settings (ii) and (iii), by Corollary 1 and  $\mathcal{O}(\log k)$  writes to control bit registers; and  $\mathcal{O}(k + \log k) = \mathcal{O}(k)$  local steps in settings (iv), by Theorem 4 and the previous argument about control bits. The subsequent store operations are in a constant time each.

The collecting operation takes  $\mathcal{O}(k)$  steps, since there is only a prefix of  $\mathcal{O}(k)$  registers corresponding to the names to be read. Finally, the algorithm uses registers corresponding to names from a prefix of length  $\mathcal{O}(k)$ , which is dominated by the number of other auxiliary registers. This is  $\mathcal{O}(k \log(N/k))$  in (i), by Theorem 1; next  $\mathcal{O}(n)$  in (ii), and  $\mathcal{O}(n \log n)$  in (iii), by Corollary 1; and finally  $\mathcal{O}(n^2)$  in (iv), by Theorem 4.  $\square$

We remark that an alternative solution to (iv) in Theorem 5 was given by Afek and Levie [4].

## 4 Lower bounds on local steps

We consider the time complexity of renaming and storing. An instance of Renaming is determined by at most the following four number parameters:  $k$ ,  $M$ ,  $N$ , and  $r$ . For some of them the time can be very small; for instance, if  $k = M$  then the original names do, so the time is  $\mathcal{O}(1)$ . On the

other side of the spectrum, if  $N = \infty$  then the step complexity of any solution is  $\Omega(k)$ . To see this, observe first that if the original names affect the actions of processes, then there is such an assignment of  $n$  original names, that in any point of an execution, if there is a choice that processes can make which is affected by the original names, then all the processes might choose similarly. In particular, if processes choose not to write, so that there is no communication among them, then all want to assign the same name. Therefore one of the processes, say,  $p$  writes at some point, let it be the first such a process. After a write, some processes learn of the value written, by reading. Observe that all have to learn, since otherwise one process among these that never learn chooses the same name as  $p$ . This argument is extended by induction to imply that the process that chooses the name as the last one had to read at least  $k - 1$  times. Such phenomena occur in many simulations, as was captured by a general lower bound of Jayanti et al. [39]. We present a general lower bound which gives an estimate on  $N$ , depending on  $M$  and  $r$ , for which the lower bound  $k - 1$  on time holds.

**Theorem 6** *Any wait-free solution of Renaming requires  $1 + \min\{k - 2, \log_{2r} \frac{N}{2M}\}$  local steps in the worst case.*

**Proof:** Consider a renaming algorithm for parameters  $k$ ,  $M$ ,  $N$  and  $r$ . Our goal is to define a set  $K$  of at most  $k$  processes and an execution of these processes with a bound on the number of local steps.

The construction is recursive and proceeds through a sequence of stages, each representing a group of concurrent reads or writes to shared registers. A *stage*  $i$  results in determining a *pool*  $P_{i+1}$  of processes eligible to be considered for stage  $i + 1$ , a *residue set*  $Q_{i+1}$ , and an initial segment  $\mathcal{E}_i$  of an execution. When the construction terminates, a subset the pool and the residue together make a set  $K$  we seek. The construction starts with the initial  $P_0$  including  $N$  conceptual processes, each identified by a specific original name, the residue  $Q_0$  being empty, and  $\mathcal{E}_0$  being an empty sequence. We may assume that any process at any event has either a read or a write to a shared register enabled.

We begin by defining the first stage, which is determined by the initial configuration. Let  $W(1)$  be the set of processes in  $P_0$  that have a write enabled in the initial configuration, and let  $R(1)$  be the set of the remaining processes in  $P_0$  that want to read from a shared register. There are two cases, depending on the relative sizes of these two sets.

The case of  $|R(1)| \geq |W(1)|$ :

There is an auxiliary register that is to be read by a group of at least  $\frac{|R(1)|}{r} \geq \frac{N}{2r}$  processes, by the pigeonhole principle. The remaining case  $|R(1)| < |W(1)|$  is symmetric: there is an auxiliary register that is to be written to by a group of at least  $\frac{|R(1)|}{r} \geq \frac{N}{2r}$  processes. Define  $P_1$  to be this group, depending on the case. Having determined  $P_1$ , we also determine an initial segment  $\mathcal{E}_1$  of the execution: it consists of the events enabled in all the processes in  $P_1$ , one event per process, these events occurring in arbitrary order.

The case of  $|R(1)| < |W(1)|$ :

This means that all the processes in  $P_1$  write. The process that writes last to the register in  $\mathcal{E}_1$  is made the only element of  $Q_1$ , otherwise  $Q_1 = Q_0$  remains empty.

We continue in a recursive way, maintaining the following invariant, for  $i \geq 0$ :

- 1) the pool  $P_i$  includes at least  $\frac{N}{(2r)^i}$  processes,
- 2) all the processes in  $P_i$  have exactly the same history of reading from the shared registers in  $\mathcal{E}_i$ ,
- 3) all the processes that wrote a value to a shared register that has ever been read in  $\mathcal{E}_i$  are in  $Q_i$ ; there are at most  $i$  such processes.

We show how to go from  $i$  to  $i + 1$ , given a pool  $P_i$  and residue  $Q_i$ , for  $i \geq 1$ . Namely, there are at least  $\frac{N/(2r)^i}{2r} = \frac{N}{(2r)^{i+1}}$  processes in  $P_i$  that have a read of the same register or a write to the same register enabled, similarly as in the first stage. We define  $P_{i+1}$  to be this group of processes, and then have the events the processes in  $P_i$  enabled occur in arbitrary order. If the events are all writes, then the last write is added to  $Q_i$  to make  $Q_{i+1}$ .

We continue for the smallest number of stages  $t$  such that the inequality  $\frac{N}{(2r)^t} \geq 2M$  holds, which determines  $t \leq \log_{2r} \frac{N}{2M}$ . This means that  $P_t$  contains at least  $2M$  elements. Simultaneously, we want to have  $t \leq k - 2$ , so that  $Q_t$  has at most  $k - 2$  elements. These two requirements combined determine  $t = \min\{k - 2, \lceil \log_{2r} \frac{N}{2M} \rceil\}$ . The definitions of  $P_t$  and  $Q_t$  imply that the processes in  $P_t \setminus Q_t$  have not written yet in  $\mathcal{E}_t$  and have read the same values from the same shared registers in the same order of reading. The size of  $P_t \setminus Q_t$  is at least  $2M - (k - 2) \geq M + 1$ .

Suppose, to arrive at a contradiction, that a decision on a new name is made by each among these  $M + 1$  processes without any further reads or writes. By the pigeonhole principle, there are two processes  $p_1$  and  $p_2$  that decide on the same name. We set  $K = Q_t \cup \{p_1, p_2\}$ , which has at most  $k$  elements. There is an execution of the algorithm, with the processes in  $K$  as the only contenders, such that  $\mathcal{E}_t$  restricted only to the events involving the processes in  $K$  is its prefix. The processes that performed writes of the values ever read are in  $K$ . Therefore  $p_1$  and  $p_2$  cannot see a difference with  $\mathcal{E}_t$  up to performing all the reads as in  $\mathcal{E}_t$ . Since the algorithm is a wait-free solution to Renaming, both  $p_0$  and  $p_1$  eventually decide on the same name, which contradicts the specification of Renaming. It follows that at least one of the processes  $p_1$  and  $p_2$  eventually performs at least one more read than in  $\mathcal{E}_t$ , resulting in a total of at least  $1 + \min\{k - 2, \log_{2r} \frac{N}{2M}\}$  reads by this process.  $\square$

**Theorem 7** *Any wait-free solution of Store&Collect requires storing to take  $\Omega(\min\{k, \log_{2r} \frac{N}{k}\})$  local steps in the worst case.*

**Proof:** The proof is similar to that of Theorem 6, with the main difference in that it is enough to continue until the stage  $\min\{k - 2, \lceil \log_{2r} \frac{N}{k} \rceil\}$ . This duration guarantees that less than  $k$  registers have been written to and there are at least  $k$  processes available to choose without violating the history of processes in the execution.  $\square$

Theorems 6 and 7 imply that both renaming and storing require  $\Omega(k)$  local steps in the worst case when the range of the original names is not known.

## 5 Unbounded selection

In this section we consider problems about continuous selection of integer values, with the goal to eventually exclusively select positive integers in such a manner that as few of them are not used.

A selected integer value may be used to identify a register to store a value, or for other purposes as an abstract “name.”

We distinguish between “storing” and “depositing” a value. A value written to a register is stored in it as long as it is not overwritten by a different value. Depositing a value means storing the value “forever” in a unique register. Repository is a concurrent data structure for depositing values in shared read-write registers. Each process may occasionally generate a value to be deposited in a repository. Registers dedicated for deposits can store any such a value. Additionally, a protocol may use auxiliary registers, with their number depending on the number  $n$  of processes. We assume that an auxiliary shared register can store one integer of arbitrary magnitude.

A complete definition is as follows. *Depositing a value  $x$  in a register  $R$*  is considered achieved at an event, when the following is satisfied:

**Storing:** The value  $x$  is stored in  $R$ .

**Persistence:** This value  $x$  will not be overwritten in register  $R$  by any different value in the remaining course of the execution.

We assume that there are infinitely many shared read-write registers  $R_1, R_2, R_3, \dots$  to store deposited values. We say that these registers are *dedicated* to depositing and that  $i$  in the *index* of  $R_i$ . All these registers are initialized to be “empty” by the `null` value. For each register  $R$  dedicated for depositing, only values attempted to be deposited are ever written into  $R$ , except for the initialization with `null`. This means that we separate such registers from auxiliary read-write registers, if any are needed. We consider it sufficient to deposit any value generated for the purpose to be deposited only once, which is considered a matter of economy rather than a restriction on implementations of depositing, so this is not made a part of specification.

The following is the repertoire of operations a process may invoke.

Operation `Depositp(v)` is invoked by a process  $p$  to deposit value  $v$ . The operation is considered completed by an acknowledgment event `ackp(R)`, where  $R$  is the register in which  $v$  has been deposited. When a process crashes while working to deposit a value, such that depositing this value has not been acknowledged yet before the crash, then the value may either get deposited or not.

Operation `Queryp` is invoked by process  $p$  to obtain a new value to be deposited. This operation is terminated by `returnp(v)`, where  $v \neq \text{null}$  is a new value to deposit, while  $v = \text{null}$  indicates that there is no value to deposit yet.

When a process  $p$  obtains a return of a query for a new value, then the process eventually invokes `Queryp` again. We assume *fair occurrence of deposit requests* at processes, which means that each process eventually obtains a new value to deposit, after having deposited the previous value, if any, unless the process crashes.

Following the generally accepted understanding of simulating executions [21], we prohibit “pipelining” on these operations in an execution. This means that a process may invoke a new operation only after the previous one, if any, has been acknowledged as completed.

The following is a complete definition. A *repository* is a concurrent data structure that allows each process to deposit values in dedicated registers subject to the following constraints on this operation:

**Persistence:** For any register  $R$  dedicated for depositing, after an acknowledgment event  $\text{ack}(R)$ , no value is ever written to  $R$ .

**Non-blocking:** Each time at least one nonfaulty process wants to deposit a value, then eventually a value gets deposited.

The Repository problem is simply to implement a repository.

We may observe that there are trivial solutions to this problem. For instance, one in which a process  $i$  deposits only in consecutive registers with indices congruent to  $i$  modulo  $n$ . A drawback of this solution is that asymptotically a small fraction of only  $1/n$  of the dedicated registers might be used to deposit values, if all but one process eventually crash. This also means that in this solution it may occur that infinitely many registers available for depositing are never used successfully to deposit a value.

On the other hand, no algorithm depositing values can guarantee that a value gets deposited in a distinguished specific register, since otherwise the value stored there could be used to determine a decision in a solution of Consensus, which is impossible [21, 33]. This means that it may happen that some registers are never used for deposits. We present solutions in which the number of dedicated registers not used for deposits is bounded above by a constant depending on the number  $n$  of processes in the system.

**Implementations of a repository.** The algorithms we give next resort to a renaming procedure. We interpret a newly acquired name  $i$  as an indication that the register  $R_i$  is available for depositing. The renaming procedure we use is similar to the wait-free solution given by Attiya et al. [14, 21].

We start from the algorithm called SELFISH-DEPOSIT, its details are as follows. Each process  $p$  maintains a sorted list  $L_p$  of  $2n - 1$  indices  $i$  of registers  $R_i$  in its local memory. This list is interpreted as storing indices of registers available for deposits. The list is initialized to store the first  $2n - 1$  positive integers arranged in order. Process  $p$  also stores a pointer  $A_p$  to the next possibly available empty register, with  $A_p$  initialized to  $2n$ .

Process  $p$  may use a procedure to *verify the list*  $L_p$ , which is performed as follows. Process  $p$  scans the list  $L_p$  and for each entry  $j$  in it reads  $R_j$ . If  $R_j$  is still empty, then the next entry  $j + 1$  is considered, otherwise  $p$  removes the  $j$ th entry from the list  $L_p$  and begins scanning registers  $R_i$  one by one in the order of indices starting from the index stored in  $A_p$ . This is continued until an empty  $R_k$  is found, if any, then  $k$  is appended to  $L_p$ . Next the entry  $j + 1$  in  $L_p$  is processed. The procedure ends with all the entries of  $L_p$  have been processed.

We will use an atomic-snapshot object  $W$ , which includes read-write registers  $W_p$ , for each process  $p$ , for  $1 \leq p \leq n$ , such that  $W_p$  is writable by  $p$  and readable by all; each  $W_p$  is initialized to be empty. These registers  $W_i$  are supported by other registers in  $W$  to provide  $W$  with the functionality of an atomic snapshot object [1, 21]. Processes  $p$  write entries from their lists to their registers  $W_p$  in  $W$ . After taking a snapshot of  $W$ , process  $p$  assigns itself *rank* defined as follows: it is the rank of  $p$  among the indices  $q$  of  $W_q$  such that  $W_q$  stores an entry of the current list  $L_p$ .

Occasionally, a process  $p$  may need to choose a value to propose as name in a special way, we call this *choosing by rank*. This is accomplished as follows. Let  $k$  be the rank of  $p$  at this point, based on the current list  $L_p$  and the most recent snapshot taken by  $p$ . Process  $p$  identifies the entry  $x$  in  $L_p$  that is of rank  $k$  among the entries that do not appear in the snapshot, and this  $x$  is the value chosen. Such a chosen value is well determined since the list  $L_p$  is long enough. Namely,



the worst case occurs when  $p = n$ , there is only one other value equal to  $W_n$  in the snapshot of  $W$ , all the remaining  $n - 2$  values are distinct, and all the values in the snapshot appear in  $L_n$ . This is because then process  $n$  needs to skip  $n - 1 + n - 1 = 2n - 2$  entries in  $L_n$  and use the very last one.

Each process  $p$  starts by querying for a new value to deposit. Once a process obtains such a value, it joins the pool of processes working to acquire a new “name” to indicate a register. The following action is iterated: process  $p$  chooses an entry in  $L_p$  and writes it into  $W_p$ . The value chosen is the first (smallest) entry in  $L_p$ , unless specified otherwise. After each write to  $W_p$ , process  $p$  takes a snapshot of  $W$ . What happens next is as follows, and is broken into two cases.

The first case is when the value  $i$  at  $W_p$  is unique in the snapshot. Then process  $p$  reads  $R_i$ : if  $R_i$  is empty then process  $p$  stores in  $R_i$  the value that needs to be deposited and acknowledges **Deposit<sub>p</sub>**, otherwise  $p$  verifies the list  $L_p$ . After the verification has been completed, if this occurs, process  $p$  again writes to  $W_p$  the smallest entry in  $L_p$  and takes a snapshot.

The second case is when process  $p$  takes a snapshot and the value at  $W_p$  is not unique. Let  $k$  be the rank of  $p$  at this point. Process  $p$  chooses by rank an entry from the list  $L_p$ , writes this entry into  $W_p$ , and takes a snapshot of  $W$ .

**Theorem 8** *Depositing based on algorithm SELFISH-DEPOSIT is a non-blocking implementation of a repository such that at most  $n - 1$  dedicated deposit registers are never used for depositing.*

**Proof:** Once a value  $x$  gets stored in  $R_i$  by a process  $p$ , then this value will not be overwritten. To see this, observe that before a write, process  $p$  takes a snapshot with  $i$  unique in it and then additionally verifies  $R_i$  if it is empty. Process  $p$  may in the future propose other names, by which  $i$  is erased in  $W_p$ , but this occurs only when  $R_i$  already stores the deposited value. A possible problem is that  $i$  could occur multiple times uniquely in a snapshot, but only the first would result in a deposit, since the next ones would be prevented by the processes double checking on  $R_i$  for emptiness.

Next we argue, by contradiction, that there are infinitely many deposits. Suppose there is an event after which there will be no deposits. Observe that all the non-faulty processes keep verifying the lists and eventually all their lists are equal and store the indices of the smallest empty deposit registers. The values on this list make a space of possible names of the size of  $2n - 1$ . Let us take the first event when this occurs. Consider the first event afterwards when each non-faulty process  $p$  has written at least one entry from its list  $L_p$  to  $W_p$ . Since this moment the ranks of all the processes become fixed. Consider the subsequent writes to  $W_p$  by process  $p$ . If such a write does not produce a unique name in the view, then the next write is after choosing by rank. Observe that once the ranks become fixed, each choosing by rank produces a unique entry in the common list shared by all the processes. Eventually such an entry is proposed as name, verified in the view to be unique, and the respective deposit register is checked to be empty, which results in a deposit.

When a process chooses by rank an entry from its list, then there is a repetition of the currently proposed names in the view, so the process skips at most  $n - 1$  entries in the list to identify the outcome of choosing. In the worst case, some of  $n - 1$  entries may happen to be skipped in each such an instance in the execution, so that none of them is ever utilized for a deposit.  $\square$

**Corollary 2** *Algorithm SELFISH-DEPOSIT minimizes the number of unused dedicated registers among non-blocking implementations of a repository.*

**Proof:** We argue that in each implementation of a repository, at least  $n - 1$  dedicated registers may be never used for deposits in some execution. Namely, when a process  $p$  is to deposit by writing to a register  $R$ , and a write event to store the value is enabled, we may “freeze” the write. At this point, no other process  $q$  will want to deposit to  $R$ , because otherwise after  $\text{ack}_q(R)$  occurs, the pending write of  $p$  to store at  $R$  may be unfrozen, which results in overwriting  $R$  in contradiction of the definition of a repository. This means that when  $p$  crashes rather than being merely “frozen,” then the register  $R$  is never used for depositing by any other process. Up to  $n - 1$  crashes can happen, so at least these many registers might never be used for deposits. This needs to be combined with the fact that at most  $n - 1$  dedicated registers are not used for depositing when algorithm SELFISH-DEPOSIT is executed, by Theorem 8.  $\square$

Next we consider a wait-free implementation of a repository. It is provided by an algorithm we call ALTRUISTIC-DEPOSIT, which is an extension of SELFISH-DEPOSIT. The algorithm is specified as follows.

There is an  $n \times n$  array  $\text{Help}[i, j]$ , for  $1 \leq i, j \leq n$ , of shared read-write registers. The intuition is that a process  $i$  writes a name to be used by process  $j$  into  $\text{Help}[i, j]$ . The difference between the algorithms is what a process  $p$  does with acquired names. In algorithm SELFISH-DEPOSIT, the names are used selfishly as addresses of registers to deposit. In algorithm ALTRUISTIC-DEPOSIT, the names are shared in the following manner. Each process  $p$  keeps reading  $\text{Help}[p, *]$  in a cyclic manner. If some  $\text{Help}[p, q]$  is  $\text{null}$  then  $p$  attempts to obtain a new name  $x$ . When the name is successfully acquired,  $p$  writes it into  $\text{Help}[p, q]$ . When a process  $p$  needs to deposit, it keeps reading  $\text{Help}[*, p]$  in a cycle. When a name  $x \neq \text{null}$  is found at  $\text{Help}[r, p]$ ,  $p$  deposits in  $R_x$  and writes  $\text{null}$  to overwrite value  $x$  in  $\text{Help}[r, p]$ . Each process  $p$  is simultaneously running the operation of scanning  $\text{Help}[p, *]$  looking for  $\text{null}$  to replace by a name, and of scanning  $\text{Help}[*, p]$  looking for a name to deposit. These operations are fairly interleaved, for instance, process  $p$  invokes the events from them in an alternating manner.

**Theorem 9** *Depositing based on algorithm ALTRUISTIC-DEPOSIT is a wait-free implementation of a repository such that at most  $n(n - 1)$  dedicated deposit registers are never used for depositing.*

**Proof:** Consider an event  $e$  in which a process  $q$  wants to deposit a value. Since SELFISH-DEPOSIT is non-blocking, there are arbitrarily many instances of processes acquiring new names after  $e$ . Eventually some process  $p$  will read the whole row  $\text{Help}[p, *]$ , and if  $\text{null}$  is encountered, then it is replaced by a name. This means that  $\text{Help}[p, q]$  contains a name. Eventually  $q$  will read  $\text{Help}[p, q]$  and use the address stored there to deposit.

Once a name  $x$  gets written into  $\text{Help}[p, q]$ , only  $q$  may use it to deposit a value into  $R_x$  and next erase the name. The worst case of waste of names occurs when at some point each entry of  $\text{Help}[*, *]$  stores a name, but then  $n - 1$  processes crash and the remaining process, say,  $p$  uses only the names in  $\text{Help}[*, p]$ .  $\square$

The optimality of  $n(n - 1)$  as an upper bound on the number of registers not used for depositing, among wait-free implementations of a repository, is open.

**Unbounded selection of names.** Next consider the problem when names are to be accumulated by processes in an exclusive manner. We call this Unbounded-Naming problem and is specified as follows.

A nonnegative integer  $i$  is considered to be “assigned to process  $p$ ” when  $p$  commits to this integer as its consecutive name by entering a special state while  $i$  is stored in a dedicated local memory variable. This is similar to committing to a decision in solutions of Consensus. A difference is that after committing to a value, and reserving it thereby, next values can be committed to. We want to have as many positive integers  $i$  to be committed to in this fashion, while no two processes ever commit to the same integer  $i$ . This goal is achievable in such a manner that a bounded set of integers is left as being not assigned to any process. We show how to adapt protocols for depositing to achieve such a goal.

The protocols for deposits we developed have processes use newly acquired names as indices of registers. When a new name  $i$  is acquired and confirmed by a snapshot of  $W$ , and hence stored at this point in  $W$ , the process reads  $R_i$  before depositing: this is to check if possibly this value  $i$  has been previously assigned as a name but later erased in  $W$ . Such a mechanism is not available for abstract assigning of names without any direct record made in shared registers, because we want to have finitely many auxiliary registers only.

Each process  $p$  has a suite of shared registers  $B_p$  in which it stores the entries of the list  $L_i$  and the value of pointer  $A_p$ . Let there be  $2n$  such registers in  $B_p$ . This information indicates which integers are still *available for name according to  $p$* : these are the numbers on  $L_p$  and numbers at least as large as  $A_p$ . When a process  $p$  acquires a new name as justified by a snapshot of  $W$ , this is not sufficient to commit to  $i$  as a name. Instead,  $p$  reads all the shared registers  $B_q$  for all the processes  $q$  to verify if  $i$  is available for a name.

We may observe that while  $i$  is in the snapshot, no other process will claim  $i$  as a name. Therefore if some process knows that  $i$  is not available for naming, this is a record of some activity that occurred before  $p$  took the snapshot. When  $p$  verifies that all processes believe  $i$  is still available,  $p$  commits to  $i$  as its next name and removes  $i$  from  $L_p$  and updates the shared registers accordingly, and after that may overwrite  $i$  in  $W_p$  at any time.

This is a general mechanism that can be applied to each of the two algorithms for deposits we gave. This shows the following fact:

**Theorem 10** *The Unbounded-Naming problem can be solved by  $n$  processes in a non-blocking fashion by an algorithm that leaves at most  $n - 1$  nonnegative integers not assigned as names, or in a wait-free manner by an algorithm that leaves at most  $n(n - 1)$  integers never assigned.*

It is an open problem if it may always happen that  $n(n - 1)$  integers are not assigned in any wait-free solution to Unbounded-Naming with  $n$  processes.

## References

- [1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.
- [2] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 91–103, 1999.

- [3] Y. Afek, P. Boxer, and D. Touitou. Bounds on the shared memory requirements for long-lived adaptive objects (extended abstract). In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 81–89, 2000.
- [4] Y. Afek and Y. D. Levie. Efficient adaptive collect algorithms. *Distributed Computing*, 20(3):221–238, 2007.
- [5] Y. Afek and M. Merritt. Fast, wait-free  $(2k - 1)$ -renaming. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 105–112, 1999.
- [6] Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive collect with applications. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 262–272, 1999.
- [7] Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.
- [8] D. Alistarh. The renaming problem: Recent developments and open questions. *Bulletin of EATCS*, 117:102–141, 2015.
- [9] D. Alistarh, J. Aspnes, K. Censor-Hillel, S. Gilbert, and R. Guerraoui. Tight bounds for asynchronous renaming. *Journal of the ACM*, 61(3):18:1–18:51, 2014.
- [10] D. Alistarh, J. Aspnes, K. Censor-Hillel, S. Gilbert, and M. Zadimoghaddam. Optimal-time adaptive strong renaming, with applications to counting. In *Proceedings of the 30th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 239–248, 2011.
- [11] D. Alistarh, J. Aspnes, G. Giakkoupis, and P. Woelfel. Randomized loose renaming in  $O(\log \log n)$  time. In *Proceedings of the 32nd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 200–209, 2013.
- [12] D. Alistarh, H. Attiya, S. Gilbert, A. Giurgiu, and R. Guerraoui. Fast randomized test-and-set and renaming. In *Proceedings of the 24th International Symposium on Distributed Computing (DISC)*, volume 6343 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2010.
- [13] J. Aspnes, G. Shah, and J. Shah. Wait-free consensus with infinite arrivals. In *Proceedings on the 34th ACM Symposium on Theory of Computing (STOC)*, pages 524–533, 2002.
- [14] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.
- [15] H. Attiya, F. E. Fich, and Y. Kaplan. Lower bounds for adaptive collect and related objects. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 60–69, 2004.
- [16] H. Attiya and A. Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM Journal on Computing*, 31(2):642–664, 2001.
- [17] H. Attiya and A. Fouren. Algorithms adapting to point contention. *Journal of the ACM*, 50(4):444–468, 2003.

- [18] H. Attiya, A. Fouren, and E. Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.
- [19] H. Attiya, F. Kuhn, C. G. Plaxton, M. Wattenhofer, and R. Wattenhofer. Efficient adaptive collect using randomization. *Distributed Computing*, 18(3):179–188, 2006.
- [20] H. Attiya and S. Rajsbaum. The combinatorial structure of wait-free solvable tasks. *SIAM J. Comput.*, 31(4):1286–1313, 2002.
- [21] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley, 2nd edition, 2004.
- [22] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *Proceedings of the 12th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 41–51, 1993.
- [23] A. Brodsky, F. Ellen, and P. Woelfel. Fully-adaptive algorithms for long-lived renaming. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, volume 4167 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2006.
- [24] H. Buhrman, J. A. Garay, J. Hoepman, and M. Moir. Long-lived renaming made fast. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 194–203, 1995.
- [25] J. E. Burns and G. L. Peterson. The ambiguity of choosing. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 145–157, 1989.
- [26] M. R. Capalbo, O. Reingold, S. P. Vadhan, and A. Wigderson. Randomness conductors and constant-degree lossless expanders. In *Proceedings of the 34th ACM Symposium on Theory of Computing (STOC)*, pages 659–668, 2002.
- [27] A. Castañeda, M. Herlihy, and S. Rajsbaum. An equivariance theorem with applications to renaming. *Algorithmica*, 70(2):171–194, 2014.
- [28] A. Castañeda and S. Rajsbaum. New combinatorial topology bounds for renaming: the lower bound. *Distributed Computing*, 22(5-6):287–301, 2010.
- [29] A. Castañeda and S. Rajsbaum. New combinatorial topology bounds for renaming: The upper bound. *Journal of the ACM*, 59(1):3, 2012.
- [30] B. S. Chlebus and D. R. Kowalski. Asynchronous exclusive selection. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 375–384, 2008.
- [31] B. S. Chlebus, D. R. Kowalski, and A. A. Shvartsman. Collective asynchronous reading with polylogarithmic worst-case overhead. In *Proceedings of the 36th ACM Symposium on Theory of Computing (STOC)*, pages 321–330, 2004.
- [32] W. Eberly, L. Higham, and J. Warpechowska-Gruca. Long-lived, fast, waitfree renaming with optimal name space and high throughput. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*, volume 1499 of *Lecture Notes in Computer Science*, pages 149–160. Springer, 1998.

- [33] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
- [34] E. Gafni, M. Merritt, and G. Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proceedings of the 20TH ACM Symposium on Principles of Distributed Computing (PODC)*, pages 161–169, 2001.
- [35] M. Helmi, L. Higham, and P. Woelfel. Space bounds for adaptive renaming. In *Proceedings of the 28th International Symposium on Distributed Computing (DISC)*, volume 8784 of *Lecture Notes in Computer Science*, pages 303–317. Springer, 2014.
- [36] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.
- [37] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999.
- [38] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.
- [39] P. Jayanti, K. Tan, and S. Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.
- [40] M. Merritt and G. Taubenfeld. Computing with infinitely many processes. *Information and Computation*, 233:12–31, 2013.
- [41] M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, 1995.
- [42] M. Moir and J. A. Garay. Fast, long-lived renaming improved and simplified. In *Proceeding of the 10th International Workshop on Distributed Algorithms (WDAG)*, volume 1151 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 1996.
- [43] M. E. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus - making resilient algorithms fast in practice. In *Proceedings of the 2nd ACM/SIAM Symposium on Discrete Algorithms (SODA)*, pages 351–362, 1991.